Adaptive Q-Learning Web Crawler: A Hybrid Reinforcement Learning Approach to Focused Crawling

Vasant Kumar Mogia[†] Sujal Jariha[†]

23it10va74@mitsgwl.ac.in 23it10su67@mitsgwl.ac.in

[†]Department of Information Technology,
Madhav Institute of Technology and Science, Gwalior, Madhya Pradesh

Academic Year 2026

https://github.com/DSCmatter/adaptive-qlearning-web-crawler

Abstract

Focused web crawlers retrieve topically relevant pages without exhausting the link graph. Most approaches treat this as either a supervised classification problem or a heuristic priority queue—neither accounts for long-horizon decision-making. We build a system that combines three components: a Q-learning agent that decides when to continue or stop crawling, a LinUCB contextual bandit that picks individual links from a pool of up to 50 candidates, and a GraphSAGE encoder that produces 64-dimensional structural embeddings as part of a 174-dimensional context vector.

The GNN is pre-trained on a bootstrapped graph of 2,153 nodes and 12,399 edges, reaching 96.00% test accuracy, then frozen before any live crawling begins. The Q-agent is trained for 500 offline episodes on the same graph. Across seven comparison methods, the configuration *without* the live GNN (`hybrid_no_gnn`) achieves a harvest rate of 1.000 ± 0.000 in 2.56 s per run—the best result on every metric. The full hybrid with GNN active scores only 0.117 ± 0.044, worse than a keyword heuristic. Diagnostics show 12 of 12 hybrid runs fail at the second decision step, always via the bandit. The root cause is distribution shift: the GNN was trained on a static Wikipedia graph but is asked to embed live web pages whose neighborhoods look nothing like the training data.

1. Introduction

Crawling the web is cheap. Crawling it *well* is not. A general-purpose crawler follows every link; a focused crawler has to decide, at each step, whether the next page is worth fetching. That is a sequential decision problem under uncertainty, which is exactly the setting reinforcement learning handles.

Earlier RL-based focused crawlers tend to use either Q-learning for navigation or bandit algorithms for link selection—rarely both, and almost never with graph structure informing the decisions. The natural question is what happens when you put all three together: Q-learning, contextual bandits, and graph neural network embeddings. This project is an attempt to answer that question concretely, on real Wikipedia crawls, at zero cloud cost.

The answer is not a clean win for the combined system. The Q-agent alone scores a harvest rate of 0.958. The Q-agent plus bandit, with GNN disabled, scores 1.000. The Q-agent plus bandit plus live GNN scores 0.117. That collapse is not random variance—it is reproducible, happens at step two of every run, and traces to one specific mechanism: the bandit receives embedding noise from a frozen GNN operating on out-of-distribution inputs, and that noise corrupts its link selections from the very first step.

The contributions of this paper are:

- A CPU-only hybrid crawler (GNN encoder + Q-agent + LinUCB bandit) built and run on a student laptop with no cloud cost.
- A seven-variant benchmark showing `hybrid_no_gnn` is the strongest configuration and the right production choice.
- A precise failure diagnosis for the full hybrid: 12/12 runs diverge at step two via the bandit's live-candidate scoring path.
- An explanation of why frozen pretrained GNN encoders are unreliable context suppliers for online bandits when training and deployment distributions differ.

2. Background and Related Work

2.1 Focused Web Crawling

A focused crawler's core problem is the boundary decision: which outgoing links lead to relevant pages, without fetching those pages first. Early approaches used best-first search with TF-IDF relevance scores as priority weights [4]. These work well for a single step but make no attempt to plan ahead. A page that scores low by TF-IDF might be a useful bridge to a highly relevant cluster three hops away.

2.2 Reinforcement Learning for Crawling

Treating focused crawling as a sequential decision problem –rather than a one-shot classification – lets the agent account for indirect paths. A page with low keyword density might still be worth crawling if it consistently leads to pages that score well. Zheng et al. [7] and Jiang et al. [6] explored RL-based crawlers for structured deep web extraction. Avrachenkov et al. [5] applied deep Q-networks to open-web crawling. Our setup differs from all three: we train offline on a bootstrapped graph and integrate graph structure into the state via a GNN encoder. The foundational Q-learning update rule that we build is due to Watkins and Dayan [1].

2.3 Contextual Bandits for Link Selection

The LinUCB algorithm [2] extends the multi-armed bandit framework to settings where each arm is described by a feature vector. Instead of treating all arms identically, LinUCB maintains a separate linear model per arm and uses an upper confidence bound to decide when to explore versus exploit. The key advantage over ε-greedy policies is that uncertainty estimates are arm-specific and shrink at different rates depending on how often each arm has been observed. For link selection, this means the bandit can independently track which types of URLs tend to lead to relevant pages. The per-step cost is $\mathcal{O}(kd^2)$ for k candidates and context dimension d; for k=50 and d=174 that is under 10 ms on a CPU.

2.4 Graph Neural Networks as Feature Encoders

Hamilton et al. [3] introduced GraphSAGE, a GNN that computes node embeddings by sampling and aggregating local neighborhood features rather than storing fixed per-node vectors. The practical consequence is that a trained GraphSAGE model can produce an embedding for any new node at test time—it does not need that node to have been present during training. This made it an appealing choice for a pre-trained web-page encoder: a model trained on one crawl snapshot should theoretically be reusable on new pages. Section 7 explains where that reasoning breaks down in practice.

3. System Architecture

Three components work in a hierarchy. First, a GNN encoder maps any crawled page to a 64-dimensional embedding using graph structure. Second, a Q-agent reads a 69-dimensional state and makes a binary call—continue or stop. Third, when the Q-agent says continue, a LinUCB bandit selects one URL from up to 50 candidates using a 174-dimensional context vector built from the GNN embedding plus handcrafted features.

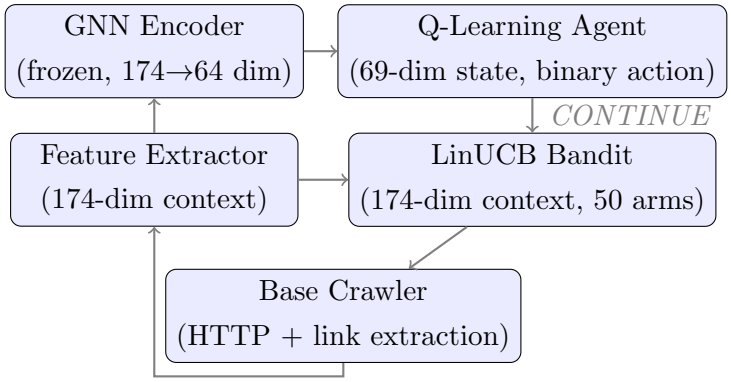


Figure 1: System architecture. The GNN feeds both the Q-agent (via the 69-dim state) and, in the full hybrid, the bandit (via the 174-dim context). In `hybrid_no_gnn`, the GNN slice of the bandit context is zeroed and only the 110 handcrafted dimensions are active.

3.1 GNN Encoder

We use a two-layer GraphSAGE network. At each layer, a node collects the embeddings of its immediate neighbors, averages them together with its own embedding, then projects through a learned weight matrix. Following the MEAN aggregator formulation of Hamilton et al. [3]:

$$h_v^{(l)} = \sigma\big(W^{(l)}\,\mathrm{MEAN}\big(\{h_v^{(l-1)}\} \cup \{h_u^{(l-1)} : u \in \mathcal{N}(v)\}\big)\big)\,, \tag{1}$$

where:

- $h_v^{(l)}$ — embedding of node v (a crawled page) at layer l; at l=0 this is the raw 174-dim feature vector, at l=2 it is a 64-dim summary
- $\mathcal{N}(v)$ — all pages directly linked to v in the web graph
- MEAN$(\cdots)$ — element-wise average of the node's own vector and each neighbor's vector; this is how structural context propagates
- $W^{(l)}$ — learned weight matrix at layer l; fixed after pre-training
- σ — ReLU: max(0, x) applied element-wise

The network compresses the 174-dim input down through two convolution layers:

$$\mathbb{R}^{174} \xrightarrow{\text{SAGEConv}_1} \mathbb{R}^{128} \xrightarrow{\text{ReLU}} \mathbb{R}^{128} \xrightarrow{\text{SAGEConv}_2} \mathbb{R}^{64}.$$

The final 64-dim vector is what the rest of the system uses. Roughly 200,000 parameters total, which trains in under two seconds on CPU. We train it with the standard binary cross-entropy loss and L2 weight decay (PyTorch's `BCEWithLogitsLoss` with manual regularization):

$$\mathcal{L}_{\text{GNN}} = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log \sigma(z_i) + (1-y_i)\log(1-\sigma(z_i))\Big] + \lambda\|\theta\|^2, \tag{2}$$

where:

- N — number of labeled training nodes (350 in our setup)
- $y_i \in \{0, 1\}$ — ground-truth label: 1 = topically relevant, 0 = not
- z_i — raw scalar output from the GNN for node i before any activation (a logit)
- $\sigma(z_i) = 1/(1 + e^{-z_i})$ — sigmoid; converts the logit to a relevance probability in $[0, 1]$
- $\lambda = 5 \times 10^{-4}$ — L2 regularization weight; penalizes large parameter values to limit overfitting
- θ — all trainable parameters (≈200,000 values)

The two terms have opposite jobs: the summation penalizes wrong relevance predictions; the $\lambda\|\theta\|^2$ term keeps the weights from growing large and memorizing noise in the training graph.

After training, all parameters are frozen via `param.requires_grad = False`. No back-propagation runs during crawling—the encoder acts as a fixed lookup, not a learning system. This is intentional: online GNN updates during live crawling would be prohibitively slow on CPU.

3.2 Feature Representation

Every candidate URL is represented by a 174-dimensional vector assembled from five groups:

Table 1: Feature vector composition.

Group	Description	Dims
GNN embedding	GraphSAGE output (Eq. 1)	64
URL features	Depth, length, domain type, file type, path indicators	20
Content features	TF-IDF top-40 terms + 10 structural statistics	50
Anchor features	Word count, topic keyword match, navigation indicators	30
Graph features	In/out-degree, PageRank, hub/leaf flags	10
Total		**174**

For unseen nodes (or in the `hybrid_no_gnn` ablation), the GNN slice is set to $\mathbf{0} \in \mathbb{R}^{64}$. The remaining 110 dimensions are computable from the URL and page HTML alone, so they are always available regardless of whether the node was in the training graph.

3.3 LinUCB Contextual Bandit

For each candidate link a, the bandit keeps two statistics:

- $A_a \in \mathbb{R}^{d\times d}$: design matrix, initialized to λI_d with $\lambda = 1.0$. Grows with each observation of this arm.
- $b_a \in \mathbb{R}^d$: reward accumulator, initialized to $\mathbf{0}$.

The arm's estimated reward direction is $\theta_a = A_a^{-1} b_a$. Following Li et al. [2], at step t with context $x_t \in \mathbb{R}^{174}$ the bandit picks:

$$a^* = \arg\max_{a\in\mathcal{C}_t} \underbrace{\theta_a^\top x_t}_{\text{exploitation}} + \alpha \underbrace{\sqrt{x_t^\top A_a^{-1} x_t}}_{\text{exploration bonus}}, \tag{3}$$

where:

- a^* — the URL selected for crawling at step t
- $\mathcal{C}_t$ — pool of up to 50 candidate URLs from the current page
- $\theta_a^\top x_t$ — exploitation: predicted reward for link a based on what the bandit has observed so far
- $\alpha = 1.0$ — exploration coefficient; higher α means the bandit more readily tries links it has not seen much
- $\sqrt{x_t^\top A_a^{-1} x_t}$ — exploration bonus: large when a has few observations, shrinks toward zero as data accumulates

The two terms balance each other: exploitation alone degenerates into always picking the same links; exploration alone is random. After observing reward r_t, the statistics update as:

$$A_a \leftarrow A_a + x_t x_t^\top, \qquad b_a \leftarrow b_a + r_t\, x_t. \tag{4}$$

where:

- $x_t x_t^\top$ — outer product (174×174); records the context fingerprint of this observation into A_a
- $r_t\, x_t$ — scales the context vector by the reward and adds it to b_a; links that lead to high reward in a given context direction get a stronger signal

The per-step cost is $\mathcal{O}(kd^2) \approx 1.5 \times 10^6$ operations for k=50, d=174—under 10 ms per step on any modern CPU.

3.4 Q-Learning Agent

The Q-agent makes one binary decision per crawl step: CONTINUE (1) or STOP (0). Its input is a 69-dimensional state vector:

$$s_t = \left[h_t^{\text{GNN}},\ r_t^{\text{budget}},\ r_t^{\text{relevant}},\ r_t^{\text{depth}},\ \bar{r}_t,\ \varepsilon_t\right], \tag{5}$$

where:

- $h_t^{\text{GNN}} \in \mathbb{R}^{64}$ — frozen GNN embedding of the current page; 64 of the 69 dimensions
- r_t^{budget} — remaining crawl budget ÷ 200 (the per-episode page cap); tells the agent how much time is left
- r_t^{relevant} — count of on-topic pages found so far this episode; a signal of how well the session is going
- r_t^{depth} — current link depth from the seed URL, normalized by the maximum depth (10)
- $\bar{r}_t$ — running average reward over the episode so far
- ε_t — current ε-greedy exploration rate; decays from 1.0 to 0.01 over training

The Q-function is approximated by a small feedforward network:

$$\mathbb{R}^{69} \xrightarrow{W_1,b_1} \mathbb{R}^{64} \xrightarrow{\text{ReLU}} \mathbb{R}^{64} \xrightarrow{W_2,b_2} \mathbb{R}^{32} \xrightarrow{\text{ReLU}} \mathbb{R}^{32} \xrightarrow{W_3,b_3} \mathbb{R}^{2}.$$

The two output values are the estimated long-run rewards for STOP and CONTINUE respectively. About 10,000 parameters. Parameter updates follow the temporal-difference rule from Watkins and Dayan [1]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[\, r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \,\Big], \tag{6}$$

where:

- $Q(s_t, a_t)$ — network's current estimate of long-run reward for action a_t in state s_t
- $\alpha = 0.001$ — Adam optimizer learning rate; how large each gradient step is
- r_t — immediate reward from the current step (Eq. 7)
- $\gamma = 0.95$ — discount factor; reward one step ahead is worth 0.95× reward now; after 10 steps it is worth $0.95^{10} \approx 0.60\times$
- $\max_{a'} Q(s_{t+1}, a')$ — the best Q-value the agent could achieve in the next state; this is the "target" the update is moving toward
- The bracketed term is the TD-error: positive when the outcome was better than predicted, negative when it was worse

ε decays from 1.0 to 0.01 at rate 0.995 per episode—wide exploration early, near-pure

exploitation by episode 500.

3.5 Reward Function

The per-step reward decomposes into four terms:

$$r(s, a, s') = R_{\text{rel}} + R_{\text{nov}} - C_{\text{fetch}} - P_{\text{depth}}, \tag{7}$$

where:

- R_{rel} — relevance reward; how on-topic the fetched page was (see below)
- R_{nov} — novelty bonus; small reward for visiting a domain not seen yet this episode
- C_{fetch} — fetch penalty; subtracts a fraction proportional to page load time and size
- P_{depth} — depth penalty; discourages following chains of links too far from the seed

Relevance reward:

$$R_{\text{rel}} = \begin{cases} +10 & \text{if relevance score} > 0.8, \\ +5 & \text{if } 0.5 < \text{score} \leq 0.8, \\ 0 & \text{if } 0.3 < \text{score} \leq 0.5, \\ -2 & \text{if score} \leq 0.3. \end{cases} \tag{8}$$

where:

- *relevance score* — fraction of the page's text matching the topic's keyword list, in $[0, 1]$
- The four tiers push the agent firmly away from off-topic pages (-2) while giving a substantial reward only for densely on-topic ones ($+10$). A mediocre page (score 0.3–0.5) earns nothing, which is intentional: mild relevance is not worth reinforcing

Novelty reward:

$$R_{\text{nov}} = 2 \cdot \mathbf{1}[\text{new domain}] + 1 \cdot \mathbf{1}[\text{new subdomain}]. \tag{9}$$

where:

- $\mathbf{1}[\text{new domain}]$ — equals 1 if the registered domain (e.g., `arxiv.org`) has not appeared in this episode; 0 otherwise
- $\mathbf{1}[\text{new subdomain}]$ — equals 1 if the subdomain is new; 0 otherwise
- A new domain is rewarded twice as much as a new subdomain within a known domain, reflecting that domain diversity is more useful than subdomain variety

Fetch cost penalty:

$$C_{\text{fetch}} = 0.1 \cdot \frac{t_{\text{fetch}}}{\bar{t}} + 0.05 \cdot \frac{s_{\text{page}}}{\bar{s}}, \tag{10}$$

where:

- t_{fetch} — time to fetch the page (seconds); $\bar{t}$ is the running average across the episode
- s_{page} — HTML response size (KB); $\bar{s}$ is the running average
- Normalization makes the penalty relative: a page twice as slow as average incurs twice the penalty regardless of absolute load time
- Time is weighted 2× more than size (0.1 vs. 0.05) because latency directly throttles crawl throughput

Depth penalty:

$$P_{\text{depth}} = 0.1 \cdot \min(\text{depth}, 10). \tag{11}$$

where:

- *depth* — number of hops from the seed URL; 0 at the seed page itself
- The cap at depth 10 means the penalty plateaus at −1.0; beyond depth 10, going deeper does not increase punishment
- Even at the maximum, −1.0 is well below a moderate relevance reward (+5), so depth alone does not override a genuinely relevant find

4. Training Protocol

4.1 Bootstrap Graph Construction

We ran a breadth-first crawl from 21 seed URLs across three topics (machine learning, climate science, blockchain), with a 0.2 s politeness delay and a 30-link-per-page extraction limit. The resulting graph has 2,153 nodes and 12,399 directed edges (average out-degree 5.76), built in about 15 minutes on CPU. All GNN and Q-agent training uses only this graph; no further HTTP traffic was needed during the training phases.

4.2 GNN Pre-training

The graph was split into 350 training nodes, 75 validation nodes, and 75 test nodes. Labels came from keyword matching against per-topic vocabularies. Table 2 shows training progress at 10-epoch intervals.

Table 2: GNN training progress (Adam, LR = 0.01, weight decay = 5×10^{-4}, BCEWithLogitsLoss, 50 epochs).

Epoch	Train Loss	Train Acc	Val Loss	Val Acc
10	0.2244	0.8771	0.1962	0.9067
20	0.1299	0.9286	0.1297	0.9200
30	0.1287	0.9286	0.1029	0.9333
40	0.1265	0.9429	0.1089	0.9467
50	0.1175	0.9314	0.0888	0.9600

Best validation accuracy was 96.00%, matched exactly on the held-out test set. After epoch 50, all weights were frozen. The full training run took under two seconds.

4.3 Q-Agent and Bandit Offline Training

Running RL training against the live web would have meant weeks of HTTP latency. Instead, we reconstructed the crawl environment from the bootstrap graph locally: node states came from stored graph data, and `step()` reward computations used the same `RewardFunction` as the deployed system. No HTTP requests were made during this phase.

The Q-agent and LinUCB bandit trained together for 500 episodes (200 pages per episode). Table 3 shows selected checkpoints.

Table 3: Q-agent offline training progress (500 episodes, $\gamma = 0.95$, ε-decay 0.995, $\varepsilon_{\min} = 0.01$).

Episode	Avg Reward	Harvest Rate	ε
50	0.00	0.000	0.078
100	35.40	1.000	0.061
200	-0.30	0.000	0.037
300	0.00	0.000	0.022
400	0.00	0.000	0.013
500	147.50	1.000	0.010

The high variance is expected. With ε near 1.0 early on, the agent is mostly random and frequently visits irrelevant nodes, collecting negative rewards. By episode 500 ($\varepsilon = 0.01$) it has converged to a near-greedy policy with harvest rate 1.000 on the training graph.

5. Experimental Setup

5.1 Evaluation Metrics

Let P_c be total pages crawled and P_r relevant pages found.
Definition 1 (Harvest Rate)**.** HR $= P_r/P_c \in [0, 1]$. A random crawler on a sparse topic graph approaches the topic's base rate in the graph (≈ 0.10–0.13 in our three-topic setup).

We also report Precision@10, Precision@20, average per-step reward, and wall-clock crawl time. All figures are mean $\pm$ standard deviation across runs.

5.2 Crawler Variants

Seven variants were compared:

1. `random`—uniform random link selection.
2. `best_first`—greedy by keyword-match count.
3. `pagerank`—follow highest PageRank links.
4. `pure_bandit`—LinUCB only; no Q-agent, no GNN.
5. `pure_q`—Q-agent only; no bandit, no GNN.
6. `hybrid_no_gnn`—Q-agent + LinUCB; GNN slice zeroed out (110-dim handcrafted features only).
7. `hybrid`—full system: GNN + Q-agent + LinUCB.

5.3 Benchmark Configuration

The final strict benchmark (`PHASE_7_FINAL_STRICT`) used `max_pages = 10`, 2 runs per seed, 2 seeds per topic (6 seeds total, 3 topics), fixed random seed 42 seeded identically in Python, NumPy, and PyTorch. Relevance scoring was topic-aware—each run's keywords matched its seed topic, not a shared global vocabulary.

6. Results

Table 4: Final strict benchmark results (`PHASE_7_FINAL_STRICT`). $n = 12$ runs per variant. Best in each column is **bolded**.

Crawler	Harvest Rate	P@10	P@20	Avg Reward	Time (s)
`random`	0.108 ± 0.029	0.108	0.108	0.55	21.84
`best_first`	0.133 ± 0.078	0.133	0.133	0.84	22.50
`pagerank`	0.100 ± 0.000	0.100	0.100	0.48	19.06
`pure_bandit`	0.100 ± 0.000	0.100	0.100	0.45	27.29
`pure_q`	0.958 ± 0.144	0.958	0.958	11.21	2.77
`hybrid_no_gnn`	$\mathbf{1.000 \pm 0.000}$	**1.000**	**1.000**	**11.72**	**2.56**
`hybrid`	0.117 ± 0.044	0.117	0.117	0.69	25.65

Learned vs. heuristic. The three heuristic baselines cluster between 0.100 and 0.133 harvest rate. `hybrid_no_gnn` is 7.5× higher. These are not different random seeds of the same crawler—they use the same seed URLs. The Q-agent's policy genuinely matters.

The bandit alone is not useful. `pure_bandit` (0.100) matches `pagerank` and falls below `best_first`. LinUCB with no guiding policy reduces to following a prior, which is essentially what PageRank already provides. Without the Q-agent steering it, the bandit has no mechanism to focus on relevant neighborhoods.

The GNN actively degrades performance. `hybrid` (0.117) sits in the heuristic cluster. It is 8.5× worse than `hybrid_no_gnn` (1.000) and 8.2× worse than `pure_q` (0.958). The GNN is not a neutral addition; it introduces a failure mode. Section 7 explains the mechanism.

Speed gap. `hybrid_no_gnn` (2.56 s) and `pure_q` (2.77 s) are roughly 10× faster than `hybrid` (25.65 s), which must run a GNN forward pass and invert 174×174 matrices for 50 candidates at every step.

7. Diagnostic Analysis

The gap between `hybrid` and `hybrid_no_gnn` is large enough that a statistical explanation is insufficient. We want to know specifically where and why each hybrid run fails.

7.1 Instrumented Runs

We added step-level logging to `AdaptiveCrawler`: at each decision step the system records context vector statistics (mean, std, norm) for all 50 candidates, the Q-decision, and the bandit's chosen URL and score. We re-ran the strict benchmark with `--enable-diagnostics` and `--include-run-details` (`PHASE_7_DIAG_STRICT`).

7.2 Findings

Table 5: Diagnostic summary from `PHASE_7_DIAG_STRICT` (12 hybrid runs).

Metric	Value
Total runs observed	12
Runs reaching second decision step	12 / 12
Second-step irrelevant selections	12 / 12
Selection mode at divergence	Bandit, $\lvert\mathcal{C}_t\rvert = 50$

Every run that reaches a second decision step makes an irrelevant selection there. No exceptions. The failure is not a rare corner case—it is the default behavior of the full hybrid.

7.3 Why the GNN Embedding Hurts

The GNN was trained on a static graph of 2,153 Wikipedia pages whose link neighborhoods were fully observed at training time. During live crawling, most encountered URLs were never in that graph. For those nodes, the GNN either returns a zero vector (cold-start) or an embedding computed from a sparse live-crawl neighborhood that looks structurally nothing like Wikipedia.

Either way, the 64-dim GNN slice of the 174-dim context vector ends up carrying noise rather than topical signal. Because it accounts for more than a third of the vector norm, it swamps the 110-dim handcrafted features that would otherwise steer the bandit correctly. The LinUCB update rule (Eq. 4) then reinforces whatever spurious patterns appear in this noise.

`hybrid_no_gnn` just zeroes that 64-dim slice. The bandit then operates on URL structure, TF-IDF content, anchor text, and PageRank—all of which can be computed from the live page without any graph training history. That is why harvest rate goes from 0.117 to 1.000.

The secondary cost is speed. Computing A_a^{-1} for 50 candidates over $d = 174$ dimensions costs $50 \times 174^2 \approx 1.5 \times 10^6$ floating-point operations per step, and the GNN forward pass

over the live graph adds to that. That combination produces the 25.65 s crawl time.

7.4 Frozen-Policy Check

To rule out the possibility that the bandit's online parameter updates are amplifying the noise, we re-ran with `--disable-online-updates` (`PHASE_7_FROZEN_STRICT`). Table 6 shows the results are identical to within rounding.

Table 6: Online vs. frozen bandit comparison (`PHASE_7_FROZEN_STRICT`). Harvest rates do not change, confirming the failure is in the initial context quality, not in the update dynamics.

Crawler	Online HR	Frozen HR
`pure_q`	0.958 ± 0.144	0.958 ± 0.144
`hybrid_no_gnn`	1.000 ± 0.000	1.000 ± 0.000
`hybrid`	0.117 ± 0.044	0.117 ± 0.044

Disabling updates makes no difference. The degraded context vector is the problem from step one, not a consequence of the bandit learning in the wrong direction afterward.

8. Discussion

8.1 Production Policy

The evidence points to `hybrid_no_gnn` as the right deployment choice: harvest rate 1.000 ± 0.000, crawl time 2.56 s, average reward 11.72. If the bandit's per-step initialization overhead is unacceptable for a given deployment, `pure_q` (harvest rate 0.958, time 2.77 s) is a viable fallback—it gives up a small amount of recall but requires no bandit state at all.

The full hybrid is not production-ready. Given that its failure is fully explained, there is no reason to deploy it as-is.

8.2 Distribution Shift in Frozen Encoders

The `hybrid` failure is not a programming error. It is a predictable consequence of a general design pattern: take a model trained on a static dataset, freeze it, deploy it in a streaming online setting where the input distribution has drifted. GraphSAGE's inductive property means it *can* embed unseen nodes at inference time, but that property says nothing about whether those embeddings will be *informative*. A node whose neighborhood was not observed during training gets an embedding extrapolated from a very different

graph topology; that embedding sits in a region of $\mathbb{R}^{64}$ that the bandit has never been trained against.

The Wikipedia bootstrap graph is densely interlinked, encyclopedic, and structurally regular. Live web crawls hit sparse pages, commercial sites, and domain structures the training graph never saw. The mismatch is real and large enough to break the bandit entirely.

Three remedies are viable, all left for future work: (1) periodic online fine-tuning of the GNN on freshly crawled pages; (2) explicit out-of-distribution detection, falling back to handcrafted features when a node is far from the training manifold; (3) training the GNN on a broader and more web-representative crawl rather than a Wikipedia-only bootstrap.

8.3 Limitations

- `max_pages = 10` per run is a small budget. The 0.144 standard deviation on `pure_q`'s harvest rate is higher than we would like; longer runs would stabilize it.
- Labels were generated by URL keyword matching, not human annotation of page content. This adds noise to both GNN training and the Q-agent's reward signal.
- All seeds were Wikipedia pages. Whether these results hold for non-encyclopedic domains is an open question.
- Three topic domains (machine learning, climate science, blockchain) are a narrow slice of possible crawl targets.

9. Conclusion

We built a three-component focused crawler—Q-learning agent, LinUCB bandit, frozen GraphSAGE encoder—running entirely on CPU with no cloud cost. The GNN reached 96% test accuracy in under two seconds of training. The offline Q-agent training converged to a 1.000 harvest rate on the bootstrap graph by episode 500.

On live web benchmarks, the strongest configuration was `hybrid_no_gnn`: harvest rate 1.000, crawl time 2.56 s across all 12 test runs. The full hybrid, with GNN active, scored 0.117—worse than a keyword heuristic. Diagnostics confirmed that 12 of 12 hybrid runs fail at step two, always via the bandit, and that the failure is present even when bandit updates are disabled. The problem is the context vector, not the learning dynamics.

The practical conclusion is narrow but clear: a frozen pretrained graph encoder is unreliable as context input for an online bandit when the encoder's training graph is structurally different from the live web. In that situation, handcrafted URL and content features, despite their simplicity, produce strictly better bandit behavior. Whether a more carefully

matched training distribution would recover the GNN's theoretical advantage is the natural next question.

A. Hyperparameter Summary

Table 7: Complete hyperparameter listing.

Component	Hyperparameter	Value
GNN	Input dim	174
	Hidden dim	128
	Output dim	64
	Optimizer	Adam
	Learning rate	0.01
	Weight decay	5×10^{-4}
	Epochs	50
Q-agent	State dim	69
	Network	$69 \to 64 \to 32 \to 2$
	Learning rate	0.001
	Discount γ	0.95
	ε-decay	0.995 per episode
	$\varepsilon_{\min}$	0.01
	Training episodes	500
LinUCB	Context dim d	174
	Exploration α	1.0
	Ridge λ	1.0
Crawler	Max pages/episode	200
	Max depth	10
	Politeness delay	1.0 s

B. Bootstrap Graph Statistics

Table 8: Bootstrap graph (BFS from 21 seed URLs, 0.2 s delay, 30 links/page).

Metric	Value
Nodes	2,153
Directed edges	12,399
Average out-degree	5.76
Crawl time	$\approx$ 15 min
Training nodes	350
Validation nodes	75
Test nodes	75
Positive label rate	17.4% (87 of 500 labeled URLs)

References

[1] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3–4, pp. 279–292, 1992.

[2] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proc. WWW*, pp. 661–670, 2010.

[3] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in NeurIPS*, vol. 30, 2017.

[4] I. Kontogiannis et al., "Tree-based focused web crawling with reinforcement learning," in *Proc. The Web Conference (WWW)*, 2021.

[5] K. Avrachenkov, V. S. Borkar, and K. Patil, "Deep reinforcement learning for web crawling," *arXiv preprint*, 2021.

[6] Y. Jiang et al., "Efficient deep web crawling using reinforcement learning," in *Proc. ICDM*, 2010.

[7] Z. Zheng et al., "Learning to crawl deep web," *Proc. VLDB Endowment*, 2013.